smartie: a Python module for processing CCP4 logfiles

Peter Briggs

CCP4, CSE Department, STFC Daresbury Laboratory, Warrington WA4 4AD

Email: p.j.briggs@dl.ac.uk

---

1 Introduction: what is smartie?

smartie is a Python module containing classes and methods intended to provide tools for processing and interrogating CCP4 log files to more easily access generic information about the log file contents - in particular, information about which CCP4 programs have been run, warning messages that were generated, and any formatted tables of data that have been written.

smartie is built around the idea of providing a high-level view of a logfile that was loosely inspired by the HTML document object model (DOM) used for example in Javascript, where the DOM provides an interface for interrogating and manipulating the content of an HTML document. Smartie's equivalent "logfile object model" is much more modest, but still provides a way to drill down into some of the details of a logfile's content. In particular, smartie offers a nice way to easily interact with data in CCP4-formatted tables. The name "smartie" reflects one of the module's origins as the possible driver for a "smart logfile browser", although realising this aim is still some way off.

This article gives an overview of the ideas behind smartie, and by way of examples shows how to use its classes and functions:

- section 2 introduces smartie's "logfile object model" and explains how smartie attempts to impose structure on log files from programs and scripts.
- section 3 shows how smartie's classes and functions can be used to explore some of the "gross features" of a log file, including processing CCP4-formatted "tables" that are read by loggraph, and is generously peppered with working code examples
- section 4 briefly overviews two applications that use some of smartie's functions.

The final sections give overviews of smartie's limitations, possible future directions, and where to get hold of the code if you've been tempted to try it out after reading this article.

2 Background: an introduction to smartie's "logfile object model"

2.1 Anatomy of a logfile

155 scala.log is a log file from a run of the CCP4i "Scale and Merge Intensities" task. Principally this task runs the Scala program, but depending on the options set in the interface by the user a number of other programs may also be run before and after Scala. This log file will be used to help illustrate some of the features of smartie, and to introduce the "logfile object model".

As human beings who are more or less familiar with the output of CCP4i and CCP4 programs, we can scan through the log file by eye and recognise many different features, for example we are able to distinguish that this particular log file contains output from CCP4i (header and tail) plus logfile output from each of the programs that were run as part of the task script. We can see that some of the programs also generated some warning messages and tables (which we could view graphically using the loggraph program), and that the program output also contains CCP4 summary tags and HTML markup. Although the file is essentially one big "blob" of text, we are nonetheless able to impose a conceptual structure on the file and thus find our way around.

Similarly, smartie tries to impose some structure onto the log file by attempting to interpret it as a sequence of **fragments** - each fragment being a distinct section of the log file that is delimited by some generic feature that smartie recognises. Examples of these generic features include program banners and termination messages (such as for example those shown in figure 1), and particular forms of messaging from CCP4i.

(a)
```
##################################################################
##################################################################
##################################################################
### CCP4 6.0: SORTMTZ            version 6.0       : 06/09/05##
##################################################################
User: pjx  Run date: 13/ 7/2007 Run time: 17:09:59

Please reference: Collaborative Computational Project, Number 4. 1994.
"The CCP4 Suite: Programs for Protein Crystallography". Acta Cryst. D50, 760-763.
as well as any specific reference in the program write-up.
```

(b)
```
 SORTMTZ:  Normal termination
Times: User:       0.1s System:    0.0s Elapsed:     0:00
```

Figure 1: example of a CCP4 program banner (a) and termination message (b).

After identifying a fragment, smartie tries to make its identification more specific. If a fragment has features that match the output of a CCP4 program then classifies this as a **program**; if it looks like output from CCP4i then this is classified as **CCP4i information**. It is also possible that a fragment cannot be classified further - in this case smartie just keeps this as a fragment.

For a program fragment, smartie examines the features that it recognises and tries to extract some additional (generic) information - for example the program name and version, the user who ran it, the date and time and so on. It also extracts information about any keywords or file openings that are reported in standard formats, example of which are shown in figure 2.

(a)

```
Data line--- title Example run with aucn data
Data line--- name project DMSO crystal DMSO dataset red_aucn
Data line--- exclude EMAX     10.0
Data line--- partials     check     test 0.95 1.05     nogap
Data line--- intensities PROFILE     PARTIALS
Data line--- final PARTIALS
Data line--- scales     rotation SPACING 5     secondary 6     bfactor ON     BROTATION SPACING 20
Data line--- UNFIX V
Data line--- FIX A0
Data line--- UNFIX A1
Data line--- initial MEAN
Data line--- tie surface 0.001
Data line--- tie bfactor 0.3
Data line--- cycles 10 converge 0.3 reject 2
Data line--- output AVERAGE
Data line--- print brief nooverlap
Data line--- RSIZE 80
```

(b)

```
OPENED INPUT MTZ FILE
Logical Name: HKLIN   Filename: /home/pjx/CCP4_REMOTE/ccp4/examples/toxd/toxd.mtz
```

Figure 2: example of keyword echoing (a) and file opening report (b) from a CCP4 program logfile.

Log files can also contain formatted **tables** that are marked up in the CCP4 "$TABLE" format, and which can be interpreted by programs such as loggraph in order to produce graphs. Similarly the log file may also contain warning and informational messages **keytext messages** marked up in the CCP4 "$TEXT" format. (Keytext messages are typically produced by calls within CCP4 programs to either the Fortran library subroutine CCPERR, or the C library function ccperror - see the *CCP4LIB* documentation at http://www.ccp4.ac.uk/dist/html/ccplib.html or the *ccp4_general.c File Reference* at http://www.ccp4.ac.uk/dist/html/C_library/ccp4__general_8c.html for more information.)

Both these types of feature are described in the CCP4 program documentation, for example at http://www.ccp4.ac.uk/dist/html/loggraphformat.html, and examples of each are also shown in figure 3.

```
$TABLE: Rfactor analysis, stats vs cycle  :
$GRAPHS:<Rfactor> vs cycle :N:1,2,3:
:FOM vs cycle :N:1,4:
:-LLG vs cycle :N:1,5:
:Geometry vs cycle:N:1,6,7,8:
$$
      Ncyc   Rfact   Rfree     FOM         LLG  rmsBOND  rmsANGLE rmsCHIRAL $$
$$
         0   0.206   0.178   0.889     16260.1    0.030    3.367    0.248
         1   0.189   0.196   0.868     15382.6    0.022    2.213    0.136
         2   0.181   0.204   0.856     15228.2    0.022    2.060    0.122
         3   0.177   0.210   0.849     15206.6    0.022    2.048    0.121
         4   0.175   0.215   0.843     15211.9    0.022    2.044    0.122
         5   0.174   0.221   0.839     15215.6    0.022    2.044    0.124
         6   0.174   0.222   0.837     15214.7    0.022    2.039    0.126
         7   0.174   0.225   0.835     15220.9    0.022    2.036    0.127
         8   0.173   0.227   0.833     15230.7    0.022    2.030    0.128
         9   0.173   0.228   0.832     15234.0    0.022    2.028    0.130
        10   0.172   0.229   0.831     15233.9    0.022    2.030    0.131
$$
```

(a)

```
$TEXT:Warning: $$ comment $$
WARNING:  **** Beware! - Cell dimensions could permit Twinning ****
$$
```

(b)

Figure 3: examples of tabular data formatted into the CCP4 "$TABLE" markup for loggraph (a), and an informational message marked up in the "$TEXT" format (b).

Typically both tables and keytext messages are generated by programs, however in principle they can fall anywhere in the log file and so do not themselves represent specific fragments of the file. Instead they are considered to be a part of the fragment that encloses them in the file.

Finally: logfiles from CCP4 programs can also contain "summary tags" and HTML formatting. An example logfile chunk containing examples of both is shown in figure 4. Smartie keeps track of the start and end summary tags in the log file as a whole but doesn't (yet) assign them to specific fragments. (HTML formatting is unfortunately more of a nuisance than anything to smartie and so is generally ignored.)

```
<B><FONT COLOR="#FF0000"><!--SUMMARY_BEGIN-->

    WILSON PLOT for Ranges   35 -  60
    Resolution range:  3.9460  3.0127
<pre>
  LSQ Line Gradient =         -48.178608
 Uncertainty in Gradient  =      0.2197E+01
  X Intercept             =     -0.4395E+01
 Uncertainty in Intercept =      0.9699E-01

 For a wilson plot          B          =  -  gradient
                          SCALE        = exp( - intercept).

 Least squares straight line gives:   B  = 48.179        SCALE  =  81.04675
    where  F(absolute)**2 = SCALE*F(observed)**2*EXP(-B*2*SINTH**2/L**2)

<!--SUMMARY_END--></FONT></B>
```

Figure 4: example of a section of logfile enclosed in "summary tags", specifically <!--SUMMARY_BEGIN--> and <!--SUMMARY_END-->, and also containing additional HTML markup.

2.2 Introducing smartie's classes for the logfile object model

Once smartie has processed a log file, it is represented by a **logfile object** (this object and the others introduced here are described in more detail in section 3). The individual sections of the original file (fragments, programs and CCP4i info blocks) are also each represented by distinct objects, with the objects describing programs and CCP4i info blocks being derived from the generic fragment object.

The logfile object holds a master list of all the fragments that smartie recognised, in the order that they were encountered in the file. It also keeps a smaller list holding the subset of fragments that are programs, and another list of the subset that were CCP4i info blocks. Tables, keytexts and summaries are represented by specialised objects that are not derived from any of the other classes of objects. The logfile object keeps a master lists of all the tables, all the keytexts and all the summaries found in the file. In addition each fragment also has its own lists of the tables and keytexts that were found in that fragment.

To get a summary printout of smartie's analysis of the 155_scala.log file, you can run the command:

```
> python smartie.py 155_scala.log
```

which will generate the summary output shown in figure 5.

The summary gives an overview of what smartie found in the log file. First it lists the "fragments" that it found - that is, all the discrete blocks of output. In this case, all the fragments are also program logs, so the list of "programs" that follows is the same. Examination of the `scala.exam` script shows that it does indeed run the four programs listed.

For the list of programs, the summary of each individual program log includes some of the information that has been extracted from the program banner and termination messages. It also lists the names of each of the loggraph-formatted tables within each program log file, and information on the keytext messages found (for example, a warning from TRUNCATE about the possibility of twinning).

After the program list there are also lists of all the keytext messages and tables found in the log file as a whole, plus any CCP4i messages that were found (in this case none). If the logfile had been generated from a run of a CCP4i task then it's likely that there would have been additional messages reported from CCP4i.

This example is intended to give a basic idea of how smartie represents a log file. The following sections cover using the Python classes and functions to examine log file content in more detail.

```
Running test on logparser code
command line: ['/home/pjx/PROGRAMS/smartie/smartie.py', '155_scala.log']
**** Parsing file "155_scala.log"
Summary for 155_scala.log

This is a CCP4i logfile

4 logfile fragments

Fragments:
        CCP4i info
        Program: SORTMTZ
        Program: Scala
        Program: TRUNCATE

3 program logfiles

Programs:
        SORTMTZ v6.0    (CCP4 6.0)
        Terminated with: Normal termination

        Scala   v6.0    (CCP4 6.0)
        Terminated with: ** Normal termination **

                Tables:
                Table: ">>> Scales v rotation range, red_aucn"
                Table: "Analysis against Batch, red_aucn"
                Table: "Analysis against resolution , red_aucn"
                Table: "Analysis against intensity, red_aucn"
                Table: "Completeness, multiplicity, Rmeas v. resolution,
red_aucn"
                Table: "Correlations within dataset, red_aucn"
                Table: "Run     1, standard deviation v. Intensity, red_aucn"

        TRUNCATE        v6.0    (CCP4 6.0)
        Terminated with: Normal termination

                Keytext messages:
                Warning: "WARNING:  **** Beware! - Cell dimensions could permit
Twinning ****"
                Warning: "WARNING:  ***Beware-serious ANISOTROPY; data analyses
may be invalid ***"

                Tables:
                Table: "Wilson Plot - Suggested Bfactor 52.3"
                Table: "Acentric Moments of E for k = 1,3,4,6,8"
                Table: "Centric Moments of E for k = 1,3,4,6,8"
                Table: "Cumulative intensity distribution"
                Table: "Amplitude analysis against resolution"
                Table: "Anisotropy analysis (FALLOFF).  Example run with aucn
data"


CCP4i messages in file:
        CCP4i info: "Sorting input MTZ file
/home/pjx/PROJECTS/myProject/aucn.mtz"

Tables in file:
        Table: ">>> Scales v rotation range, red_aucn" (6 rows)
        Table: "Analysis against Batch, red_aucn" (6 rows)
        Table: "Analysis against resolution , red_aucn" (10 rows)
        Table: "Analysis against intensity, red_aucn" (13 rows)
```

```
        Table: "Completeness, multiplicity, Rmeas v. resolution, red_aucn" (10
rows)
        Table: "Correlations within dataset, red_aucn" (10 rows)
        Table: "Run     1, standard deviation v. Intensity, red_aucn" (13 rows)
        Table: "Wilson Plot - Suggested Bfactor 52.3" (60 rows)
        Table: "Acentric Moments of E for k = 1,3,4,6,8" (60 rows)
        Table: "Centric Moments of E for k = 1,3,4,6,8" (60 rows)
        Table: "Cumulative intensity distribution" (11 rows)
        Table: "Amplitude analysis against resolution" (60 rows)
        Table: "Anisotropy analysis (FALLOFF).  Example run with aucn data" (60
rows)

Keytext messages in file:
        Warning: "WARNING:  **** Beware! - Cell dimensions could permit Twinning
****"
        Warning: "WARNING:  ***Beware-serious ANISOTROPY; data analyses may be
invalid ***"


Time: 1.37
```

Figure 5: summary output from smartie after processing the log file from the standard CCP4 Scala example script

3 Using smartie to look at CCP4 log files

The following sections introduce smartie's features by example, with a set of code fragments and resulting output. If you're familiar with the Python programming language then it should be simple to start using smartie and the examples should be straightforward to follow. If you're not familiar with Python then there are lots of resources at www.python.org to help you get started.

3.1 Getting started with smartie

The logfile object lies at the heart of smartie. To create a new logfile object from a from a log file (say the same `155_scala.log` that was used in the previous section), we would use the following Python code:

```
>>> import smartie
>>> logfile = smartie.parselog("155_scala.log")
```

which will create and populate a new logfile object. Once we have this object, we can use its methods to probe the content of the file - for example, to generate the same summary as shown in figure 5 we can use smartie's "summarise" function:

```
>>> smartie.summarise(logfile)
```

The following sections give some more interesting examples that illustrate the kind of information that smartie can tell you about a log file.

3.2 Examining the structure of the log file

Once a smartie logfile object has been created it is very easy to get data about its structure and content. To begin with, you might be interested in the fragments that smartie found. To look up the total number of fragments:

```
>>> logfile.nfragments()
4
```

A fragment is any particular chunk of log file that smartie recognised, and is represented by a fragment or fragment-based object. To acquire the object representing a particular fragment, use the logfile object's "fragment" method - this example returns the fragment object for the first fragment:

```
>>> fragment = logfile.fragment(0)
>>> print fragment
<smartie.ccp4i_info instance at 0x2abed92749e0>
```

Only certain types of fragment may be of interest for a particular application. The "isccp4i_info" and "isprogram" methods of the fragment-based objects can be used to find out about the object types:

```
>>>
>>> logfile.fragment(0).isccp4i_info()
True
>>> logfile.fragment(0).isprogram()
False
>>> logfile.fragment(1).isccp4i_info()
False
>>> logfile.fragment(1).isprogram()
True
```

Examination of the 155_scala.log file shows that this is as we might expect - the log file starts with a preamble message from the CCP4i script before running the first program. If we were curious about what message CCP4i actually wrote then we could look this up:

```
>>> logfile.fragment(0).message
'Sorting input MTZ file /home/pjx/PROJECTS/myProject/aucn.mtz'
```

However it's more likely that you're interested in the programs that ran. While you could figure out which fragments are program logs by examining each fragment as shown above, smartie shortcuts this by providing a methods in the logfile object specifically for examining the programs. These are examined in more detail in the next section.

3.3 Getting information about the program log output

The logfile object introduced in the previous section provides the "nprograms" method that returns the number of program fragments found in the log file, for example for 155_scala.log:

```
>>> logfile.nprograms()
3
```

This is what we should expect - there is log file output from three programs in the 155_scala.log example file.

We can get some information about the individual program fragments. We use the "program" method returns the object representing a specific program fragment, and then we can access its data via its attributes and methods. For example:

```
>>> program = logfile.program(0)
>>> program.name
'SORTMTZ'
>>> program.version
'6.0'
>>> program.termination_message
'Normal termination'
```

The attributes are populated using data that has been extracted from the program banners and termination messages in the logfile. For CCP4 programs there are a number of available data items. To get a list of all the available attributes, use the program's "attributes" method - for example:

```
>>> program.attributes()
['termination_name', 'systemtime', 'startline', 'rundate', 'name',
'usertime', '
banner_text', 'elapsedtime', 'termination_text', 'source_file', 'user',
'version
', 'date', 'ccp4version', 'endline', 'runtime', 'nlines',
'termination_message']
```

Each of these attributes is described in more detail in the module documentation for smartie. (Note that the "attributes" method is available for any fragment-like object, although the attributes themselves can vary even between different instances of the same class.)

As described in the previous section, program log file fragments can also contain formatted tables and "keytext" warning messages (in fact this is true of any kind of fragment).

```
>>> logfile.program(1).name
'Scala'
>>> logfile.program(1).ntables()
7
>>> logfile.program(2).name
'TRUNCATE'
>>> logfile.program(2).nkeytexts()
2
```

Basically this is telling us that there were 7 tables in the logfile for the second program run (Scala) and 2 keytext warnings for the third (Truncate).

Working with tables is covered in more detail in the next section, as this is currently probably the single most useful aspect of smartie's functionality. The keytexts are much simpler; the program object has a "keytext" method that returns a keytext object, and the keytext's attributes can be retrieved. For example, the first warning in the Truncate log output in the file looks like:

```
$TEXT:Warning: $$ comment $$
WARNING:  **** Beware! - Cell dimensions could permit Twinning ****
$$
```

To access this data in smartie:

```
>>> keytext = logfile.program(2).keytext(0)
>>> keytext.message()
'WARNING:  **** Beware! - Cell dimensions could permit Twinning ****'
```

The program objects also have lists of any keywords and file opening operations that were reported in the logfile, and this data can be accessed using the "keywords", "logicalnames" and "logicalnamefile" methods.

To get a list of the reported keywords for the first program in the 155_scala.log example:

```
>>> logfile.program(0).name
'SORTMTZ'
>>> logfile.program(0).keywords()
['ASCEND', 'H K L M/ISYM BATCH']
```

Each keyworded input line is returned as an item in the list; smartie's "tokeniser" function could be used to further process each line in order to break it up into discrete tokens (see section 3.6.2).

File opening reports in CCP4 log files look like this example:

```
Logical Name: HKLOUT   Filename: /tmp/pjx/PROJECT_155_4_mtz_red_aucn.tmp
```

A "logical name" (in this case "HKLOUT") is associated with a physical filename when the program is run (more information about logical names and filenames can be found in the CCP4 manual, chapter 3 section 3.2.1 "Command line arguments/file connexion").

Here is an example of getting a list of the logical names found in a particular program log, using the "logicalnames" method:

```
>>> logfile.program(1).name
'Scala'
>>> logfile.program(1).logicalnames()
['HKLIN', 'HKLOUT', 'SYMINFO']
```

The file name associated with any logical name can then be retrieved using the "logicalnamefile" method:

```
>>> logfile.program(1).logicalnamefile("HKLIN")
'/home/pjx/PROJECTS/myProject/aucn_sorted.mtz'
```

Finally, the full text of a program fragment (or any other fragment) can be retrieved from the source file using the "retrieve" method. For example, the following example would retrieve the text of the Sortmtz log output from 155_scala.log:

```
>>> print logfile.program(0).retrieve()
 ###############################################################
 ###############################################################
 ###############################################################
 ### CCP4 6.0: SORTMTZ             version 6.0       : 06/09/05##
 ###############################################################
 User: pjx  Run date: 13/ 7/2007 Run time: 17:09:59

 ...
```

Smartie's "strip_logfile_html" function is useful if you wish to remove any HTML tags in the text whilst preserving features such as CCP4 formatted tables (see section 3.6.1).

3.4 Working with tables in log files

Smartie's table objects represent CCP4 formatted $TABLES and provide a variety of methods that can be used to extract data from these tables and reformat them for output.

To look at working with the tables we'll introduce a new example log file, 156_refmac5.log. This example contains the log file from a CCP4i task run of the refinement program Refmac5. To start examining this log file using smartie we must make a new logfile object:

```
>>> import smartie
>>> logfile = smartie.parselog("156_refmac5.log")
```

Often after running Refmac5 you are interested in seeing the contents of the final table that the program writes out (the "Rfactor analysis, stats vs cycle" table) since this gives details of how the refinement behaved over each cycle and can give an idea of the quality of the result after running the program. For 156_refmac5.log the table looks like this:

```
$TABLE: Rfactor analysis, stats vs cycle  :
$GRAPHS:<Rfactor> vs cycle :N:1,2,3:
:FOM vs cycle :N:1,4:
:-LLG vs cycle :N:1,5:
:Geometry vs cycle:N:1,6,7,8:
$$
      Ncyc   Rfact   Rfree     FOM          LLG  rmsBOND  rmsANGLE rmsCHIRAL $$
$$
         0   0.228   0.221    0.851       88700.2    0.028    4.533    0.163
         1   0.184   0.201    0.872       85078.6    0.029    2.509    0.178
         2   0.169   0.195    0.878       83708.6    0.030    2.448    0.194
         3   0.162   0.193    0.879       83089.2    0.031    2.431    0.200
         4   0.158   0.192    0.879       82787.4    0.031    2.404    0.202
         5   0.155   0.190    0.879       82619.6    0.031    2.378    0.202
 $$
```

To illustrate working with smartie's table object we will show how smartie can be used to easily get information from this table - we will find out the number of cycles of refinement and the initial and final values of the R factor and of R_{free}.

First, we need to locate the table object in the logfile, since there are many tables in the file (and all of these are in the Refmac log itself):

```
>>> logfile.ntables()
7
>>> logfile.nprograms()
1
>>> logfile.program(0).name
'Refmac_5.3.0040'
>>> logfile.program(0).ntables()
7
```

(Aside: note that the same methods for tables - for example "ntables" - that are available in logfile objects are also available in fragment and program objects. So although the remaining examples only show the methods of the program objects, they can be used from the other objects in exactly the same way.)

Since we know the title of the table we're looking for, we can use the program object's "tables" method to try and look it up. The "tables" method takes a title (or a fragment of a title) and returns a list of table objects that match. For example, the table we're interested in is called "Rfactor analysis, stats vs cycle":

```
>>> tables = logfile.program(0).tables("Rfactor analysis")
>>> len(tables)
1
>>> table = tables[0]
>>> table.title()
'Rfactor analysis, stats vs cycle'
```

The search for the partial title returns a list of all matching table objects - in this case there is only one (but in the general case there could be more than one, or none). The "title" method of the table object returns the full title of the table.

Having acquired the table object, we can then get some of the data values from the columns. For example, if we wanted a list of the values in the "Ncyc", "Rfact" or "Rfree" columns, then we would use the "col" method of the table object to return the list of values in each:

```
>>> table.col("Ncyc")
[0, 1, 2, 3, 4, 5]
>>> table.col("Rfact")
[0.22800000000000001, 0.184, 0.16900000000000001, 0.16200000000000001,
0.158, 0.155]
>>> table.col("Rfree")
[0.221, 0.20100000000000001, 0.19500000000000001, 0.193, 0.192, 0.19]
```

(Note that the string values in the original table have been converted by smartie to floating point values.)

Since the "col" method returns a normal Python list, the results can be interrogated using normal methods of accessing list items - for example, to get the last value in the "Ncyc" column (which corresponds to the total number of cycles of refinement) we can use the standard Python idiom for acquiring the last element in a list:

```
>>> table.col("Ncyc")[-1]
5
```

Something else that we might wish to do is to calculate the change in the R factor and R_{free} values over the course of the refinement:

```
>>> delta_Rfact = table.col("Rfact")[-1] - table.col("Rfact")[0]
>>> print str(delta_Rfact)
-0.073
>>> delta_Rfree = table.col("Rfree")[-1] - table.col("Rfree")[0]
>>> print str(delta_Rfree)
-0.031
```

It would be straightforward to write a small Python program that combined these code fragments to return the change in R factors given any log file from Refmac5.

The table object also offers a set of methods for outputting the table in different formats: "show" prints a basic text representation without any of the $TABLE markup for loggraph, "loggraph" generates the table including the $TABLE markup, and "html" generates an HTML table that is suitable for inclusion in a webpage. The results of these three methods are shown in figure 6 below.

(a)

Ncyc	Rfact	Rfree	FOM	LLG	rmsBOND	rmsANGLE	rmsCHIRAL
0	0.228	0.221	0.851	88700.2	0.028	4.533	0.163
1	0.184	0.201	0.872	85078.6	0.029	2.509	0.178
2	0.169	0.195	0.878	83708.6	0.03	2.448	0.194
3	0.162	0.193	0.879	83089.2	0.031	2.431	0.2
4	0.158	0.192	0.879	82787.4	0.031	2.404	0.202
5	0.155	0.19	0.879	82619.6	0.031	2.378	0.202

(b)

```
$TABLE: Rfactor analysis, stats vs cycle:
$GRAPHS
 :<Rfactor> vs cycle :N:1,2,3:
 :FOM vs cycle :N:1,4:
 :-LLG vs cycle :N:1,5:
 :Geometry vs cycle:N:1,6,7,8:
$$
  Ncyc   Rfact   Rfree    FOM      LLG   rmsBOND   rmsANGLE   rmsCHIRAL $$
$$
     0   0.228   0.221   0.851  88700.2     0.028      4.533       0.163
     1   0.184   0.201   0.872  85078.6     0.029      2.509       0.178
     2   0.169   0.195   0.878  83708.6      0.03      2.448       0.194
     3   0.162   0.193   0.879  83089.2     0.031      2.431         0.2
     4   0.158   0.192   0.879  82787.4     0.031      2.404       0.202
     5   0.155    0.19   0.879  82619.6     0.031      2.378       0.202
$$
```

(c)

Ncyc	Rfact	Rfree	FOM	LLG	rmsBOND	rmsANGLE	rmsCHIRAL
0	0.228	0.221	0.851	88700.2	0.028	4.533	0.163
1	0.184	0.201	0.872	85078.6	0.029	2.509	0.178
2	0.169	0.195	0.878	83708.6	0.03	2.448	0.194
3	0.162	0.193	0.879	83089.2	0.031	2.431	0.2
4	0.158	0.192	0.879	82787.4	0.031	2.404	0.202
5	0.155	0.19	0.879	82619.6	0.031	2.378	0.202

Figure 6: examples of outputing the table data using different table object output methods
(a) Using table.show() gives a basic text printout.
(b) Using table.loggraph() regenerates the table with CCP4 $TABLE loggraph markup
(c) using table.html() generates a HTML marked-up table that can be incorporated into a webpage and viewed in a web browser.

There is also a "jloggraph" method which generates the same output as the "loggraph" method, but with additional HTML tags that allow the table to be viewed in the JLogGraph Java applet when loaded into a web page (with the caveat that the currently distributed JLogGraph code doesn't always work with smartie-generated tables - this will be fixed for future releases of JLogGraph).

3.5 Working with summary tags in logfiles

As mentioned in the overview of log files earlier in this article, smartie also keeps track of the start and end summary tags (<--SUMMARY_BEGIN--> and <--SUMMARY_END--> respectively) found in the log file. This information is stored in a list of "summary" objects stored in the log file.

To return to the 155_scala.log example file used earlier on, the number of summaries can be obtained using the "nsummaries" method of the logfile object:

```
>>> logfile.nsummaries()
32
```

Individual summary objects can be fetched from the logfile object using the "summary" method, although these objects aren't particularly interesting in themselves: the most useful thing you can do with them is invoke their "retrieve" method to fetch the actual text enclosed in the summary tags.

As an example, to fetch the 3rd summary object from 155_scala.log:

```
>>> logfile.summary(2).retrieve()
'<'B><FONT COLOR="#FF0000"><!--SUMMARY_BEGIN-->\n<li><a
href="#commandSORTMTZ">Co
mmand Input</a>\n<<li><a href="#inputSORTMTZ">Input File
Details</a>\n<li><a href
="#outputSORTMTZ">Output File Details</a>\n<!--SUMMARY_END--
></FONT></B>\n'
```

This isn't particularly pretty, however smartie's "strip_logfile_html" function (see section 3.6) can be used to clean it up a bit:

```
>>> smartie.strip_logfile_html(logfile.summary(2).retrieve())
'Command Input\nInput File Details\nOutput File Details\n\n'
```

One obvious application of this would be to write out all the summaries from a log file in one go, for example:

```
>>> for i in range(0,logfile.nsummaries()):
...     print smartie.strip_logfile_html(logfile.summary(i).retrieve())
...
```

3.6 Some other useful functions in Smartie

Smartie has a number of supporting functions, most of which are not really very interesting outside of the module. However two of the functions may be of use to other applications: "strip_logfile_html" and "tokenise", and so are described in more detail in the following sections.

3.6.1 smartie.strip_logfile_html(): cleaning up logfiles

HTML mark up was added to a number of CCP4 programs several years ago, which allowed log files from those programs to be viewed in a web browser. Within smartie however the HTML can sometimes be something of a nuisance, and so the strip_logfile_html function can be used to remove it from arbitrary text. As an example, here is a fragment of text from 155_scala.log containing HTML markup being run through the function:

```
>>> print text
<a name="commandsScala"><h3> Input keyworded commands (click for
documentation):</h3></a>
<a href="/home/pjx/CCP4I_uWORKSHOP/ccp4/html/scala.html#title">TITLE</a>
 Example run with aucn data
<a
href="/home/pjx/CCP4I_uWORKSHOP/ccp4/html/scala.html#name">NAME</a>

>>> print smartie.strip_logfile_html(text)
 Input keyworded commands (click for documentation):
TITLE
 Example run with aucn data
NAME
```

Where strip_logfile_html is most useful however is in dealing with $TABLES that have been marked up with the Jloggraph applet code. In these cases the entire table is printed inside an attribute of a "param" tag, and this function is careful to extract the text table in this case.

3.6.2 smartie.tokenise(): CCP4-style line tokeniser

The tokenise function in smartie tries to replicate the basic functionality of the keyword parser in the CCP4 libraries. The naive solution to tokenising these lines (i.e. breaking them up into discrete tokens) is to split on white space; however this approach fails for quoted tokens which themselves contain whitespace, as shown in the example below:

```
>>> print line
HKLIN "C:\Documents and Settings\pjb93\My Documents\rnase.mtz"
>>> line.split()
['HKLIN', '"C:\\Documents', 'and', 'Settings\\pjb93\\My',
'Documents\\rnase.mtz"']
>>> print smartie.tokenise(line)
```

```
['HKLIN', '"C:\\Documents and Settings\\pjb93\\My Documents\\rnase.mtz"']
```

Smartie's tokeniser function deals with this in a fashion that is more consistent with the core CCP4 suite, and may be useful in some cases.

4 Applications currently using smartie

4.1 starKey

starKey is a small program that has been developed in order to produce a summary of the programs used by jobs run within CCP4i, and was the original motivator for developing smartie. Although the CCP4i job database stores information on each task that was run in a project, it doesn't hold information on the specific underlying programs that were used in each run. So starKey uses smartie to discover this information *a posteri* from the logfiles alone.

starKey has been developed as part of CCP4's contribution to the BIOXHIT project; more information can be found at http://www.ccp4.ac.uk/projects/bioxhit.html, and starKey (amongst other software) can be downloaded from http://www.ccp4.ac.uk/projects/bioxhit_public/

4.2 MrBUMP

MrBUMP is an automated system for performing molecular replacement. Part of its operation involves running CCP4 programs such as Refmac5 and analysing the output to determine whether a putative molecular replacement solution is worth pursuing. Smartie is used within MrBUMP to extract data from the formatted tables, and in some instances to reformat the tables for output in a master log file.

MrBUMP can be obtained from the website at http://www.ccp4.ac.uk/MrBUMP/

5 Limitations and Possible Future Development

The development of smartie so far has been somewhat haphazard and there are many places where functionality is either missing or is not particularly well implemented. In particular the classes and functions for handling tables could be greatly improved (for example, in the course of writing this article it became apparent that there is no way to list the names of columns in a table).

However possibly the most serious limitation at present is that smartie cannot recognise a wider range of "non-generic" features from specific programs - for example, information that is output only from Scala or Refmac5. Capturing this specific kind of information would be a lot of work and it is not clear that it is a practical goal; however some experiments in doing this are taking place. If the data could be captured then smartie could realistically be used as the engine for a new generation of CCP4 log file browser.

Another limitation is that smartie doesn't currently recognise output from programs that don't have standard CCP4 start or end banners (an exception to this is that it does already recognise the banners from SHELXC, D and E). Some experimental code is already extant that investigates overcoming these two limitations but is not generally available, however if you are interested then it can be provided on request.

Aside from these, there are a number of possible directions that future development of smartie could take, for example the "real-time" processing of log files, or the provision of additional methods for rendering the table objects graphically for example via Tk. However since much of the development so far has been driven by functionality requests from end users, this development model is likely to continue as the preferred way forward. Therefore I would welcome any requests from potential or actual users of smartie about how the code should be developed.

6 Conclusion

smartie is a Python module containing classes and methods for performing simple processing of CCP4 log files in order to extract basic information. Smartie provides functions to break down the structure of a log file to identify the programs that were run and to access information that has been marked up in special formats such as CCP4's $TABLE tags.

In this article I have given a practical overview of how smartie's classes and functions can be used to obtain information on the "gross features" within a CCP4 log file, along with examples of working code that will hopefully allow people to see how smartie might be used in their own applications.

7 Availability

The current version of smartie at the time of writing is 0.0.9. The latest version of smartie can always be downloaded via FTP from ftp://ftp.ccp4.ac.uk/pjx/ccp4/smartie/

The distribution currently includes the smartie.py module plus a number of example logfiles that can be used to try out the code on.

Finally, if you are interested in using smartie, or have suggestions about it could be developed or used in future, then I would be very interested in working with you - please contact me at **p.j.briggs@dl.ac.uk**.

8 Acknowledgements

PJB would like to acknowledge funding from STFC Daresbury Laboratory via the CCP4 project and from the BIOXHIT project, which is funded by the European Commission within its FP6 Programme, under the thematic area "Life sciences, genomics and biotechnology for health", contract number LHSG-CT-2003-503420.

PJB would also like to acknowledge the valuable testing performed by Wanjuan Yang, and the very valuable suggestions from Ronan Keegan which were very helpful in developing smartie and in particular the table handling classes.

Finally, the figures in this article were prepared using the Gimp and the "convert" program from the Imagemagick suite.